Technical Report: Learning with a Sparse Distributed Memory

Jim Marshall
Lisa Meeden
Computer Science Department
Indiana University
Bloomington, Indiana

1 May 1989

Abstract

In this paper we describe a software implementation of a sparse distributed memory in the *Lisp language on a Connection Machine. We use several different learning tasks to evaluate the performance of the memory model, including generalization, prediction, and retraining.

1 Introduction

A sparse distributed memory (SDM) is a model of human long-term memory developed by Kanerva that may offer a new approach to problems in machine learning [Kanerva, 1988]. The memory supports approximate addressing, which means that an item stored in the memory can be retrieved by addressing the memory with an address sufficiently similar, but not necessarily identical, to the original address with which the item was written. This property of the memory allows a learning system to be designed around it which deals with the problem of pattern prediction. The system learns to predict future patterns of a sequence by matching incoming patterns against similar patterns of the sequence, which it has previously encountered and stored in its sparse distributed memory. These patterns are long binary strings, typically having length from 100 to 10,000 bits, and are used as addresses into the memory itself.

The approximate addressing mechanism of the memory causes similar patterns to address similar regions of the memory space. In processing a sequence of patterns, the learning system stores each pattern it encounters in the region of memory addressed by the preceding pattern of the sequence. Reading from the memory using a particular sequence pattern as the address retrieves the pattern which follows that pattern in the sequence. More generally, reading with a pattern which is similar, but not identical to some pattern in the sequence retrieves the pattern following the sequence pattern which most closely resembles the address pattern.

The ability of a pattern to be retrieved using an address which is not identical to the original address with which it was written depends on several factors, one of which is the number of times the pattern has been written to memory. Unlike conventional computer memories, writing new data in a sparse distributed memory does not overwrite old data, but rather increases the total amount of data in memory. The more times a pattern is written into memory at a particular address, the more dissimilar the read addresses can be from the original write address in order for the pattern to be successfully retrieved. In other words, to successfully retrieve the pattern, the address used to read from memory need not resemble the pattern's original write address as closely.

A learning system based on a sparse distributed memory can be designed as follows. Sensory input to the system is encoded as long patterns of bits, corresponding to the presence or absence of certain abstract "features" of the perceived environment. This sensory data feeds into a central region of the system called the "focus", which itself serves as the interface to and from the memory. Intuitively, a sequence of patterns in the focus can be interpreted as the subjective experience of the system over time. At any given moment, the current pattern in the focus will trigger similar patterns stored in the memory, which in turn feed back into the focus. A sequence of patterns coming in from the environment will recall similar sequences which the system has previously experienced (if any), and this allows the system to predict the outcome of its current situation. The real outcome (as perceived by the senses) can then be compared to the predicted outcome; a good match will strengthen the sequence stored in memory, whereas a poor match will cause the information stored in memory to be revised to more accurately reflect the perceived outcome.

In this paper we describe a software implementation of such a learning system, as well as several learning tasks which we have used to evaluate the performance of the system. The system currently is implemented in the *Lisp language and runs on a Connection Machine model CM-2. The following section describes the SDM model itself.

2 The Sparse Distributed Memory Model

The SDM model is based on the mathematical properties of the space of n-bit binary strings, where n is large (in the range 100 to 10,000). Both the addresses to the memory and the data stored in the memory are n-bit binary strings. This implies an enormously large address space, far too large to be physically realizable in hardware if all 2^n memory addresses have physical storage locations associated with them. In the actual model, only a very sparse subset of all possible memory addresses have physical storage locations, termed "hard locations", associated with them. These hard locations are chosen as a random sample, with replacement, from the 2^n total possible memory addresses.

The *distance* between two memory addresses is defined as the Hamming distance between the addresses, or the number of bits in which the addresses differ. Approximate memory addressing is accomplished by associating a *response region* with each hard location address. A hard location is activated during a read or write operation whenever an address is used that falls within its response region, defined to be the set of addresses within a specified Hamming distance of the hard location address. Memory read and write operations are thus distributed over many hard locations.

Writing a pattern to memory stores a copy of the pattern in each activated hard location. Data are not overwritten as in conventional computer memories; instead, the new data pattern is simply added to the existing set of data stored in the hard location. Reading the memory at a particular address retrieves all of the data patterns stored in all of the hard locations activated by the read operation. These data patterns are combined into a single statistical representative of all of the patterns, which is then returned as the result of the read operation.

A sequence of patterns can be stored in the memory as a simple linked-list data structure, whereby any given pattern is stored at the address corresponding to the pattern preceding it in the sequence. An important concept that arises as a result of the mathematical properties of the memory is that of *convergent* and *divergent* sequences. Two sequences S_1 and S_2 are said to converge if reading S_1 with pattern p_1 and S_2 with pattern p_2 in general yields two patterns whose distance is smaller than the distance between p_1 and p_2. Otherwise the sequences are said to diverge.

Whether two sequences converge or diverge depends on the distance between the initial two patterns used to read the sequences. If the initial patterns are beyond a certain distance apart, called the *critical distance*, the resulting read sequences will in general diverge. The critical distance depends on several factors, such as the size of the stored bit patterns, the number of hard locations, and the size of a hard location response region. Note that since the behavior of the memory is statistical, the critical distance is only a probabilistic indication as to whether or not two sequences will eventually converge: beyond the critical distance, divergence becomes more likely than convergence. A related phenomenon is *chance convergence*, whereby two initially divergent sequences can by chance converge to a common point or cycle of random noise. However, chance convergence tends to occur only for small memories and non-uniform data sets.

The technique of storing the successor of a pattern at the address corresponding to the pattern itself makes it easy to implement a method of predicting patterns that depend on only one previous pattern. However, such a method breaks down when the next pattern in a sequence depends not only on the previous pattern, but also on patterns two or more steps back in the sequence. This problem is remedied by extending the model to support several *folds*. A fold is a component of the memory that stores sequence transitions of a particular length. The memory as described above is simply a one-fold memory, since successive patterns depend only on patterns one step back in a sequence. A k-fold memory allows prediction of successive patterns based on the previous k patterns of the sequence.

For a more detailed description of the sparse distributed memory model, the reader is referred to [Kanerva, 1988]. The remainder of this paper describes our implementation of a k-fold sparse distributed memory system, and the learning tasks which we attempted.

3 Implementation

3.1 Memory Patterns

In order to allow easy visualization of the operation of the memory, we chose to display data patterns as 7×16 grids of characters, with '@' standing for a 1-bit and '.' standing for a 0-bit. Thus, our patterns were 112 bits long. Each pattern was chosen to resemble a visual depiction of a letter of the alphabet, as shown in Figure 1 below. Note that the addresses themselves are not meant to correspond to actual representations of visual perception data, but were merely chosen as an aid to observing the behavior of the memory. All 26 letters of the alphabet were represented in this way.

```
....@@@@@@@@@...     .@@@@@@@@@@@....     @@@@@......@@@@@
..@@@@.....@@@..     .@@@.......@@@..     @@@@@@....@@@@@@
..@@@@@@........     .@@@........@@@.     @@@.@@@..@@@.@@@
....@@@@@@@@....     .@@@........@@@.     @@@..@@@@@@..@@@
........@@@@@@..     .@@@........@@@.     @@@...@@@@...@@@
..@@@.....@@@@..     .@@@.......@@@..     @@@....@@....@@@
...@@@@@@@@@....     .@@@@@@@@@@@....     @@@..........@@@
```

Figure 1: Example memory address/data patterns

3.2 Hard Locations

Our learning system was implemented on a Connection Machine (CM) parallel computer in the *Lisp language, an extension of Common Lisp providing parallel data structures and data operations. The CM consists of an array of up to (depending on the physical size of the machine) 65,536 parallel processors configurable by the user. The machine we used consisted of 32,768 physical processors, although our memory simulations never required more than 8,192 processors.

The user may specify how many physical processors are needed for the duration of the program. Once these processors have been allocated to the user, he or she may restrict the set of active processors during execution of the program to be various subsets of the total allocated processors.

Parallel operations on data are accomplished by storing the data in the active processor set (one data item per processor), and then performing a single (parallel) data operation on all of the active processors simultaneously.

Out of the 2^{112} total possible memory addresses, we chose 8,192 addresses at random to serve as hard locations, although there is no reason why one needs to limit the model to this number. Each hard location is represented by a CM processor, so configuring the number of processors is equivalent to choosing how many hard locations will be used during a run. Each hard location (processor) contains two arrays of length 112: a bit-vector representing the hard location's address; and an array of integer counters, representing the set of all data stored in the hard location so far, about which more will be said below. The hard location address vectors and counter arrays are CM parallel variables, and essentially comprise the entire (one-fold) sparse distributed memory data structure.

3.3 Read and Write Operations

To perform a parallel write operation with approximate addressing, the write address is loaded into each processor and simultaneously compared to each internal hard location address. If the Hamming distance between the two addresses is within a predefined *access radius*, that hard location is included in the set of hard locations active during the write operation. All active hard locations store a copy of the pattern being written to memory.

It is not necessary to explicitly store each data pattern in a hard location. All that is required is an array of *bit-counters*, which store the relative numbers of 0-bits and 1-bits occurring in each bit position of the location, over the set of all data words stored in the location so far. A positive value $+v$ in counter j indicates that v more 1's have occurred in bit position j than 0's, over all the data stored in that location. A negative value indicates the amount of excess 0-bits occurring in bit position j, over all the data stored in that location.

To perform a parallel read operation, a set of active hard locations sufficiently similar to the read address is computed in a way analogous to the write operation, as described above. To compute a representative pattern from the data stored in all of the active hard locations, the bit-counters of all active locations are summed, bit-wise, to produce an array containing, for each bit position, the total excess 0-bits or 1-bits over all of the data from all of the active locations. These values are then thresholded at zero, to produce the final representative string. A positive bit-sum value corresponds to a 1-bit, a negative value corresponds to a 0-bit, and a zero value is randomly chosen to be either a 0-bit or a 1-bit. The variance of the bit-sums (i.e., the magnitudes of their absolute values) gives an indication of the relative strength of the data being read, where large variances correspond to stored data and small variances correspond to random noise. The variance values provide an internal check on whether or not stored data is being read.

Since the read and write operations are performed in parallel across all hard locations, their execution time is essentially independent of the number of hard locations. Therefore, the model scales up well.

3.4 The Focus and k-Fold Prediction

The center of the learning system is the *focus*, which provides the interface between the environment, represented as an input file containing patterns, and the other major component of the system, a k-fold sparse distributed memory, essentially just a collection of one-fold memories. The (k-fold) focus is implemented as an array of k bit-vectors, each of length 112. Each fold associates patterns of a sequence with patterns encountered several steps back in the sequence, the number of previous steps depending on the fold number. A k-fold memory allows predictions to be made based on all the information made available during the last k steps of the sequence.

At any particular time t, the k-fold memory is used to predict the outcome of the current situation, that is, the expected environment pattern at time $t+1$. Each fold of the memory is addressed with the corresponding fold of the focus. The k resulting patterns are then averaged into a final pattern (using the thresholding technique as described for reading above), which becomes the *prediction* of the memory at time $t+1$. The prediction is then combined with the input pattern read from the environment (called the *perception*), and the resulting pattern (called the *composite*) is stored back into each memory fold at each respective fold address. Finally, the composite replaces the most recent pattern of the focus (stored in fold 0); the remaining focus patterns are shifted over one position; and the oldest focus pattern (stored in fold k-1) is discarded. The various methods used to produce the composite pattern from the prediction and the perception will be described in the section on Learning below.

4 Learning

4.1 Generalization

The basic generalization task was to give the memory several different fuzzy examples of a pattern and then evaluate how well it was able to generalize from the common features. We used a single fold memory and did not employ the learning model described earlier because this model is designed to store sequences. Instead we generated noisy versions of a particular pattern and then used each noisy pattern as both the data and address in a write operation. To obtain and evaluate a generalization, we read from the memory with varying degrees of noise and calculated the Hamming distance between the memory's response and the initial clean pattern.

We wrote with patterns containing 20% and 30%random noise, which in our representation equated to 22 bits and 34 bits respectively of the 112 length bit strings. In Figure 2 below, the first grid is the initial A-pattern, and the second grid contains 20% noise, and the third 30% noise. We also varied the number of different noisy patterns which were presented to the

```
.......@@.......     ....@@@@@@......     @.......@.@.@..@
.....@@@@@@.....     ....@@@.@@@.@...     ..@...@@@@@.....
....@@@..@@@....     .....@@....@....     ....@.@..@@@...@
...@@@....@@@...     ...@@@....@@@@..     ...@@@.@@.@@@...
..@@@@@@@@@@@@..     ...@@@@@@.@..@..     ...@@.@@@@.@.@.@
.@@@........@@@.     ..@@.....@@@@@..     @.@@@.......@@..
@@@..........@@@     @@@...@@.....@@@     @@@..@........@@
   A-pattern            20% Noise            30% Noise
```

Figure 2: Noisy A-patterns

memory.

The data in table 1 summarize the memory's performance on several generalization tasks as described above. First, consider the section of the table pertaining to trials written with 20% noise. For example, with only 5 trials of noisy A-patterns, the memory returned patterns that were less than 7.5 bits away from the original A-pattern when it was read with an A-pattern of 10% noise or less. The memory's performance was even more impressive when it had more trials to generalize from. After 15 trials, when the memory was read with 10% noise, it returned a pattern only 1.7 bits away from the original pattern on average. It is important to remember that the actual clean pattern was never written during any of the trials.

Clearly the sparse distributed memory is very successful at generalizing when written with 20% noise, but not nearly as good when written with 30% noise. This drop in performance seems intuitively correct, but can also be explained by examining the memory model more closely. Each noisy pattern, in the 20% noise case, is 22 bits away from the clean pattern, but how far away are the noisy patterns from each other? Conceivably they could be anywhere from 0-44 bits away, although it is unlikely that they would be closer than 22 bits since they are generated randomly from the clean pattern. On average, the range appears to be 30-40 bits. But in the 30% noise case, the possible range is 0-68 bits, and the average range is approximately 46-56.

With our parameters, a write operation activates all hard locations within 44 bits of the write address (approximately 80 locations). Therefore, the write circle of each 20% noisy address overlaps all of the others (since they are all within 44 bits of each other), and it is in this overlapping

Written with percent noise	Number written	Read with percent noise	Average HD over 10 reads (bits)
20	5	30	14.3
		20	9.3
		10	7.5
		5	6.7
		0	4.0
	10	30	3.4
		20	3.7
		10	2.1
		5	1.6
		0	2.0
	15	30	8.3
		20	3.2
		10	1.7
		5	1.3
		0	1.0
30	5	30	25.5
		20	23.8
		10	22.5
		5	22.2
		0	21.0
	10	30	21.2
		20	20.4
		10	17.1
		5	16.5
		0	19.0
	15	30	12.9
		20	13.7
		10	11.3
		5	10.1
		0	8.0

Table 1: Summary of Generalization Results

region that the generalization is formed. But when using write data with 30% noise, the overlapping region may be significantly less, so it is more difficult for the memory to build up an adequate generalization.

The memory model also explains why the generalization improves as the memory is read with a diminishing amount of noise. When the memory is tested with a 30% noisy pattern, the generalization is not as strong because it is only pooling data from the outskirts of the overlapping region. Similarly, when it is read with less noise the generalization improves, because statistically, the center of the overlapping region should be at the locations activated by a clean pattern.

4.2 Prediction

The first prediction task we attempted was to expose a single fold memory to a sequence of patterns several times, and then to begin a read sequence in the memory starting from a fuzzy element of the original sequence. We evaluated how well the memory was able to predict the remainder of the sequence.

When given a cue with 0-15% noise, the read sequence converged to the true sequence in a single step. With a cue containing 16-22% noise, it converged in 1-2 steps, and in cues with more than 25% noise it generally diverged, which does not seem unreasonable. As with generalization, a sparse distributed memory seems well-suited for this type of learning.

However there are a few problems when learning sequences in a relatively small memory. We encountered some chance convergence to true sequences, when reading with cues containing more than 25% noise. We also converged to strange letter-like patterns that appeared to be conglomerates of the original letter patterns. In these cases the variance was quite low (below 0.7) which a learning system could monitor as an indication of the "realness" of the data. Some of the patterns in our test sequences were as close as 24 bits in Hamming distance which tended to lead to more chance convergence. If the elements of the test sequences are chosen with care, the amount of chance convergence should be quite small.

A single fold memory works well for patterns such as the alphabet where being able to predict the next letter depends only on knowing its predecessor. However, suppose we wanted to teach the memory two different words: "WARM" and "JARS". Each has the same set of middle letters "AR". So if the memory is given the "R" as a cue it must be able to predict whether an "S" or "M" should follow. It clearly cannot do this with the previous letter

```
.@@.@.@.@@@@@...     @@@.@....@@@@@@@     @@@@@......@@@@@
@@@@@@....@@@@..     @@@@@@....@@@@..     @@@@@@....@@@@.@
.@@@@@@@.@@.....     .@@@@@@@.@@...@.     .@..@@@@.@@@..@@
.@..@@@@@@@@.@..     .@...@@@@@@@.@@.     @@@..@@@@@@..@@.
@......@@@@.@@..     @@.....@@@..@@..     @@....@@@@...@@@
@.@.......@@@@..     @.@.......@@@@..     @.@....@.....@@@
.@@@..@@...@...@     .@@@..@@...@....     @@@...@....@.@@@
    Fold 0               Fold 1               Fold 2
```

Figure 3: Individual Fold Predictions

alone, or with the previous two letters, it must know whether a "J" or a "W" occurred two steps back. A 3-fold memory should be able to accomplish this task because it monitors not only 0-step transitions, but 1-step and 2-step transitions as well. A j-step transition is a pair of elements in a sequence separated by j-1 elements.

Our second prediction task was to have a 3-fold memory predict the appropriate successor to an "R" after having seen an equal number of the "JARS" and "WARM" sequences, and being currently presented with "WAR". After 3 trials of each sequence, reading with an "R" from fold 0 returned the first pattern in Figure 3. Reading with an "A" from fold 1 returned the second pattern in Figure 3. And reading with a "W" from fold 2 (the 2-step transition) yielded the third pattern in Figure 3. Only the pattern read from the last fold strongly resembles the "M" which should be predicted (the first pattern in Figure 4).

The combined prediction pattern formed from the three patterns read from each fold is the second pattern shown in Figure 4. Since each fold's response is weighted equally the combined prediction does not gain much benefit from the strong M-like response from fold 2. Fold 2's response is only 10 bits from a true "M" while the combined prediction is 36 bits away.

The third pattern in Figure 4, is the composite pattern formed from the perception and the prediction. In this particular test the composite was formed by weighting the perception with a fixed bias of 75%.

With only limited testing completed on this particular task, we have not been able to obtain a prediction any closer than 22 bits from the perception. It is not clear whether more trials will improve the performance for this task. It seems that predicting the k-step transitions in a k+1 fold memory will

```
@@@@@......@@@@@      @@@.@....@@@@@@@      @@@@@......@@@@@
@@@@@@....@@@@@@      @@@@@@....@@@@..      @@@@@@....@@@@.@
@@@.@@@..@@@.@@@      .@@@@@@@.@@...@.      @@@.@@@..@@@.@@.
@@@..@@@@@@..@@@      .@...@@@@@@@.@@.      @@@..@@@@@@..@@.
@@@...@@@@...@@@      @@.....@@@..@@..      @@@....@@@...@@@
@@@....@@....@@@      @.@.......@@@@..      @@@....@@....@@.
@@@..........@@@      .@@@..@@...@...@      @@@@...........@
   Perception            Prediction             Composite
```

Figure 4: Final Predictions After 3 WARM/JARS Trials

always be difficult because of the noise introduced from the other folds. And this cannot be remedied by simply ignoring the output from certain folds on particular time steps, because the focus in the learning model has no way of "knowing" when this would be appropriate.

4.3 Retraining

It seems reasonable to expect the memory model to be able to learn an association between two patterns, but if the first pattern in the association becomes more strongly attached to a new pattern, the memory should be able to accommodate this new association. This learning task proved to be the most difficult of all that we attempted.

Until now the specifics about how the composite pattern is formed in the focus have not been discussed, primarily because it is still not clear what method is most appropriate. The system's ability to be successfully retrained hinges on how this composite is constructed. Four different methods were used:

1. Weight the prediction according to a confidence measure.
2. Weight both the prediction and perception equally.
3. Have a fixed 75
4. Ignore the prediction entirely.

The confidence measure of the first method was based on the variance of the data read. When the variance was high, it indicated that the memory was reading stored data rather than random noise.

The system is able to successfully predict sequences using each of these methods, but its rate of learning is fastest for the first and fourth and slowest for the second.

An example of the type of task attempted is the following:

1. Initially train the system on the 2-element sequence I-pattern followed by a B-pattern (4-6 trials).

2. Introduce a new association for the I-pattern, an X-pattern and train the system on this new sequence (10-40 trials).

After only completing preliminary tests, it appears that the system is not capable of being retrained using the first or second composite-building methods. After 40 trials, reading with an I-pattern still retrieves a very strong B-pattern. For the first method, this result is due to the fact that the confidence is very high in the memory's data (since the I-B association was well-learned) so each successive composite pattern is still much more strongly weighted towards the B-pattern. For the second method, the randomness of the composite never leans strongly enough towards the X-pattern for the new association to be made.

The third method proved more successful, but still not satisfactory. After 40 trials, it was returning a pattern within 22 bits of an X-pattern when read with an I-pattern. However, it seemed to reach a plateau and failed to grow closer to the X-pattern for many ensuing time steps.

It is obvious that the final method should work, since the memory's prediction is being completely ignored. However, this method is obviously not appropriate for building a "composite" structure nor for use in a general learning algorithm.

5 Future Research

One major question to come out of our research is: In using a sparse distributed memory for learning, what is a reasonable method of reconciling incorrect predictions with perception in such a way that retraining is possible? Another central question is: Can the performance of the k-fold memory be improved when predicting the distinguishing element between two similar sequences (as in the "WARM"/"JARS" example)?

6 Acknowledgements

The authors would like to extend warm thanks to Pentti Kanerva for a stimulating discussion of his memory model. We also wish to acknowledge the use of Thinking Machine Corporation's Connection Machine Network Server link to their CM-2 system in Boston.

References

[Hillis, 1985] Hillis, W. D. (1985). *The Connection Machine*. ACM Distinguished Dissertation. MIT Press, Cambridge, Ma.

[Hillis, 1987] Hillis, W. D. (1987). The connection machine. *Scientific American*, pages 108–115.

[Kanerva, 1988] Kanerva, P. (1988). *Sparse Distributed Memory*. MIT Press, Cambridge, Ma.

[StarLisp, 1988] StarLisp (1988). *StarLisp Reference Manual*. Thinking Machines Corporations, Cambridge, MA, 5.0 edition.